0082_tsp

May 8, 2018

1 Travelling community nurse problem (aka travelling salesman problem).

The travelling salesman problem is a classic geographic problem that may be framed as "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?".

The same problem may be applied to community nurses: given a list of patients to see in a day, what order should the nurse visit the patients to minimise time spent travelling (to free up the most time for nursing time).

This problem is a class of problems known as 'np-hard', that is the problem size grows non-linearly with the number of points to visit, and soon becomes too computationally expensive to compute all possibilities. There is also no algorithm known that efficiently gives a certain optimum solution. For example with 5 points to visit there are 120 possible routes, but with 10 points there are 3,628,800 possible routes, and with 20 points to visit there are 2,432,902,008,176,640,000 routes!

Np-hard problems are 'solved' by using a 'heuristic algorithm'. Heuristic algorithms follow a method designed to efficiently give a sufficiently good solution at the expense of not guaranteeing that an optimal solution is found.

In the case of the travelling salesman problem (or, in our case, a community nurse with a set of patients to visit), there are are many heuristics described. It is a favourite problem of algorithm writers!

In the code below we will use a 'hill-climbing' method based on reversing portions of the route (or a 'pairwise exchange' approach). Hill-climbing methods perform some search that then leads to them picking the best improvement in that step of the search. The search method is then repeated until no further improvement is found. Hill-climbing methods may get trapped in local optima in complicated problems – the solution found depends on the starting point. To reduce the impact of this the algorithm is repeated multiple times with different start points.

Our algorithm will follow these steps:

1) Pick a route at random (randomly order to points to visit)

2) Examine all possible pairwise exchanges in the route in (e.g. if we have a route A-B-C-D-E-F-G-H, and pairwise exchange C and F we have a new route A-B-F-E-D-C-G-H). Choose the pairwise exchange that gives the best reduction in route distance).

3) Repeat step 2) until no further improvement

4) Repeat from step 1 for a determined number of repeats (or repeat until a maximum algorithm use time is reached).

In order to run an algorithm like this we need to know the travel times or distances between points. In the example below we calculate a straight line (or 'Euclidean') distance based on x and y locations. x/y locations are readily available for addresses (such as Eastings and Northings in the UK). Straight line distances or travel times, while only approximate may be good enough to get reasonable solutions. More precise solutions could use travel times or distances determined from Geographical Information Systems (an example of an implementation using Open Source methods and open mapping data is described here: https://github.com/MichaelAllen1966/batch_travel_times_and_distances).

In [4]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.path import Path
import matplotlib.patches as patches
import math


def main_code():
    """Main program code called"""

    # Load points to visit with co-ordinates
    points_to_visit, xCo, yCo, number_of_points_to_visit = load_points()

    # Create a lookup dictionary for distances between any two points_to_visit
    distances_lookup = create_dictionary_of_distance(
        number_of_points_to_visit, xCo, yCo)

    # Run route finding algorithm multiple times
    number_of_runs = 50
    (final_best_distance_so_far, final_best_route_so_far,
     progress_in_best_distance) = (find_shortest_route(
             number_of_runs, points_to_visit, distances_lookup))

    # Print results
    print_results(final_best_route_so_far, xCo, yCo, points_to_visit,
                  final_best_distance_so_far, progress_in_best_distance)


def calculate_total_route_distance(route, distances_lookup):
    """Calculates route distances. Accesses 'distances_lookup' dictionary which
    gives distance between any two points. Total distance includes return to
    starting point at end of route"""

    total = 0
    for i in range(len(route) - 1):
        total += distances_lookup[route[i], route[i + 1]]
    total += distances_lookup[route[-1], route[0]]
    return total
```

```
def create_dictionary_of_distance(number_of_points_to_visit, xCo, yCo):
    """Creates a dictions of distances_lookup between all combinations of
    points_to_visit"""

    distances_lookup = {}

    for i in range(number_of_points_to_visit):
        for j in range(i, number_of_points_to_visit):
            x1 = xCo[i]
            y1 = yCo[i]
            x2 = xCo[j]
            y2 = yCo[j]
            distance = math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
            distances_lookup[i, j] = distance
            distances_lookup[j, i] = distance

    return distances_lookup


def load_points():
    """Usually this function will load point ids and co-ordinates from, for
    example, a CSV file. Here we define a function that will generate random
    points, as an example"""

    number_of_points = 25
    points_to_visit = list(range(0, number_of_points))
    x = np.random.uniform(0, 200, number_of_points)
    y = np.random.uniform(0, 200, number_of_points)
    number_of_points_to_visit = len(points_to_visit)

    return points_to_visit, x, y, number_of_points_to_visit


def find_shortest_route(runs, points_to_visit, distances_lookup):
    """Main algorithm code"""

    final_best_route_so_far = []
    final_best_distance_so_far = 1e99
    progress_in_best_distance = []

    for run in range(runs):
        exchange_point_1 = 0
        exchange_point_2 = 0
        improvement_found = True
        best_route_so_far = points_to_visit.copy()
        np.random.shuffle(best_route_so_far)
        best_distance_so_far = calculate_total_route_distance(
```

```
                best_route_so_far, distances_lookup)

        while improvement_found:  # continue until no further improvement
            improvement_found = False

            # Loop through all paiwwise route section reversals
            for i in range(0, len(best_route_so_far)-1):
                for j in range(i, len(best_route_so_far)):
                    test_route = best_route_so_far.copy()
                    test_route = reverse_section(test_route, i, j)
                    test_route_distance = (calculate_total_route_distance
                                (test_route, distances_lookup))
                    if test_route_distance < best_distance_so_far:
                        exchange_point_1 = i
                        exchange_point_2 = j
                        improvement_found = True
                        best_distance_so_far = test_route_distance

            if improvement_found:
                best_route_so_far = reverse_section(
                        best_route_so_far, exchange_point_1, exchange_point_2)

        if best_distance_so_far < final_best_distance_so_far:
            final_best_distance_so_far = best_distance_so_far
            final_best_route_so_far = best_route_so_far.copy()

        progress_in_best_distance.append(final_best_distance_so_far)

    return (final_best_distance_so_far, final_best_route_so_far,
            progress_in_best_distance)


def plot_improvement(progress_in_best_distance):
    """Plot improvement over runs"""

    plt.plot(progress_in_best_distance)
    plt.xlabel('Run')
    plt.ylabel('Best distance')
    plt.show()


def plot_route(route, xCo, yCo, label):
    """Plot points and best route found between points"""

    # Create figure
    fig = plt.figure(figsize=(8, 5))

   # Plot points to vist
```

```
    ax1 = fig.add_subplot(121)
    ax1.scatter(xCo, yCo)
    texts = []
    for i, txt in enumerate(label):
        texts.append(ax1.text(xCo[i] + 1, yCo[i] + 1, txt))

    # Plot best route found between points
    verts = [None] * int(len(route) + 1)
    codes = [None] * int(len(route) + 1)
    for i in range(0, len(route)):
        verts[i] = xCo[route[i]], yCo[route[i]]
        if i == 0:
            codes[i] = Path.MOVETO
        else:
            codes[i] = Path.LINETO
    verts[len(route)] = xCo[route[0]], yCo[route[0]]
    codes[len(route)] = Path.CLOSEPOLY

    path = Path(verts, codes)

    ax2 = fig.add_subplot(122)
    patch = patches.PathPatch(path, facecolor='none', lw=0)
    ax2.add_patch(patch)
    ax2.set_xlim(min(xCo) - 10, max(xCo) + 10)
    ax2.set_ylim(min(yCo) - 10, max(yCo) + 10)

    # give the points a label
    xs, ys = zip(*verts)
    ax2.plot(xs, ys, 'x--', lw=2, color='black', ms=10)

    texts = []
    for i, txt in enumerate(label):
        texts.append(ax2.text(xCo[i] + 1, yCo[i] + 1, txt))

    # Display plot
    plt.tight_layout(pad=4)
    plt.show()
    return


def print_results(final_best_route_so_far, xCo, yCo, points_to_visit,
                  final_best_distance_so_far, progress_in_best_distance):

    """Prints results to screeen"""

    print('Best route found:')
    plot_route(final_best_route_so_far, xCo, yCo, points_to_visit)
    print('\nDistance = %.0f' % final_best_distance_so_far)
```

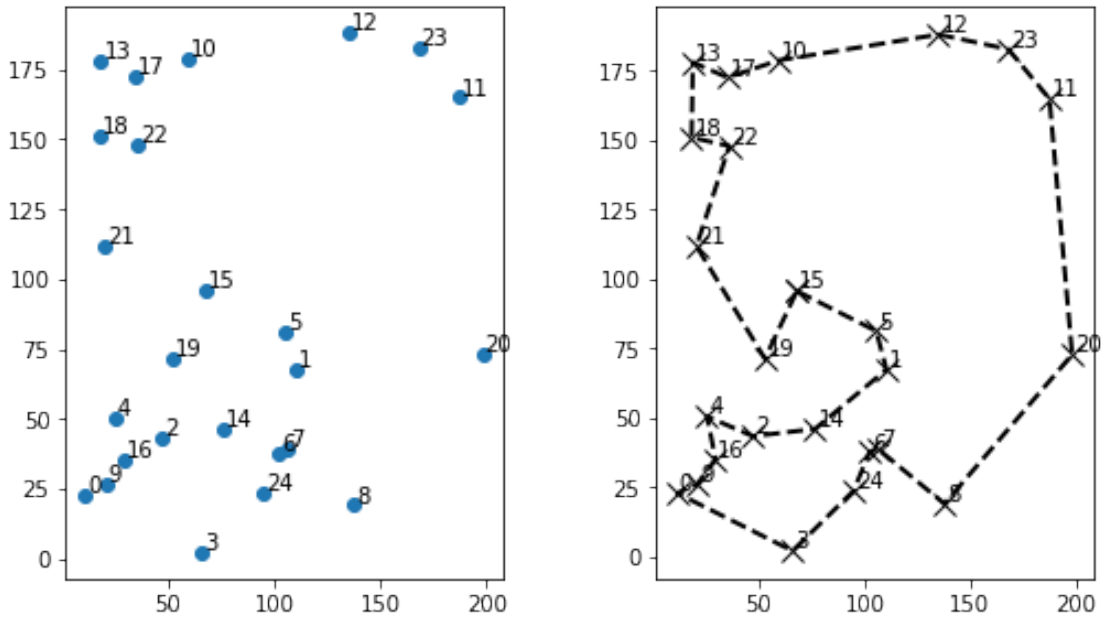
```
        print('\nImprovement in distance over run:')
        plot_improvement(progress_in_best_distance)


    def reverse_section(orig_list, point_a, point_b):
        """Reverses section of route between points a and b (inclusive)"""

        low_switch_point = min(point_a, point_b)
        high_switch_point = max(point_a, point_b)
        high_switch_point += 1  # Include high switch point in reversed section
        section_1 = orig_list[0:low_switch_point]
        section_2 = list(reversed(orig_list[low_switch_point:high_switch_point]))
        section_3 = orig_list[high_switch_point:]
        new_list = section_1 + section_2 + section_3
        return (new_list)


    # RUN MAIN CODE
    # -------------
    main_code()
```

```
Best route found:
```




```
Distance = 854
```

Improvement in distance over run: